程序员的早餐-2022/05/12 那些你可能忽视的前端性能优化细节

前端性能的好坏是影响用户体验的一个关键因素，因此进行前端相关的性能优化显得十分重要。网络上一些常见的优化手段，相信不少读者也都了解过或实践过，所以本文主要介绍一些比较容易被忽视的优化细节，当然前提都是在大规模计算的场景下。

Babel 编译优化

> 本内容运行环境为 node v14.16.0，babel 版本为 @babel/preset-env@7.17.10，benchmark 版本为 benchmark@2.1.4

众所周知 babel 有很多的配置项，不同的配置下编译出来的结果也大不相同，有些编译的结果会为了符合 ECMAScript 规范，而进行一些的额外检查或实现一些特殊的能力，从而引起一些性能上的开销，然而在多数情况下这些检查和能力带来的开销是不必要的，因此下面会列举一些常见的插件配置来进行优化。

1. @babel/plugin-proposal-object-rest-spread

在项目中有可能会使用 ... 运算符来进行进行克隆或者属性拷贝，例子如下：

```
1 const o1 = { a:1 ,b:2, c:3 };
```

```
const o2 = { x:1, y: 2, z:3 };
const o3 = { ...o1, ...o2 };
```

当使用 babel 默认配置时，该代码会编译如下代码：

```
"use strict";

function ownKeys(object, enumerableOnly) { var keys = Object.keys(object); if (Object.getOwnPropertySymbols) { var symbols = Object.getOwnPropertySymbols(object); enumerableOnly && (symbols = symbols.filter(function (sym) { return Object.getOwnPropertyDescriptor(object, sym).enumerable; })), keys.push.apply(keys, symbols); } return keys; }

function _objectSpread(target) { for (var i = 1; i < arguments.length; i++) { var source = null != arguments[i] ? arguments[i] : {}; i % 2 ? ownKeys(Object(source), !0).forEach(function (key) { _defineProperty(target, key, source[key]); }) : Object.getOwnPropertyDescriptors ? Object.defineProperties(target, Object.getOwnPropertyDescriptors(source)) : ownKeys(Object(source)).forEach(function (key) { Object.defineProperty(target, key, Object.getOwnPropertyDescriptor(source, key)); }); } return target; }

function _defineProperty(obj, key, value) { if (key in obj) { Object.defineProperty(obj, key, { value: value, enumerable: true, configurable: true, writable: true }); } else { obj[key] = value; } return obj; }

const o1 = {
  a: 1,
  b: 2,
  c: 3
};
const o2 = {
  x: 1,
  y: 2,
  z: 3
};

const o3 = _objectSpread(_objectSpread({}, o1), o2);

```

可以看出一个简单的属性拷贝里用到了很多 Object 相关的函数调用。我们用 benchmark 来测试一下属性拷贝的性能，同样的添加一组使用原生的 Object.assign 和一组使用原生 ... 扩展运算符作为对照，其代码如下：

```
const Benchmark = require('benchmark');
const suite = new Benchmark.Suite();

// ... 省略处为上方代码 3~18 行

suite
.on('complete', (event) => {
  console.log(String(event.target));
})
.add('babel _objectSpread', () => {
  var o3 = _objectSpread(_objectSpread({}, o1), o2);
}).run()
.add('Object.assign', () => {
  var o3 = Object.assign({}, o1, o2);
}).run()
.add('use ...', () => {
  var o3 = { ...o1, ...o2 };
})
.run();

```

输出的结果如下：

```
babel _objectSpread x 1,512,926 ops/sec ±0.33% (90 runs sampled)
Object.assign x 8,682,644 ops/sec ±0.33% (93 runs sampled)
use ... x 538,816 ops/sec ±1.25% (87 runs sampled)
```

可以看出使用 Object.assign 性能最优，如果项目中有大量属性拷贝的使用（特别是在一些大数据的循环中使用），那么在性能上会有很大的差距。

既然如此 babel 为什么不默认编译成使用原生的 Object.assign 进行拷贝呢，具体原因可以参考 https://2ality.com/2016/10/rest-spread-properties.html#spreading-objects-versus-object.assign() 链接中的描述，简单概况就是在对**有 Object.defineProperty 修饰过得对象**来说，其属性拷贝时存在一些小细节上的差异。

因此如果项目中不在乎上述链接中的细节差异，推荐在 babel.config.json 或 .babelrc 中添加如下配置，将其转换为使用原生的 Object.assign，配置如下：

```
  "plugins": [
    [
```

```
      "@babel/plugin-proposal-object-rest-spread",
      {
        "loose": true,
        "useBuiltIns": true
      }
    ]
  ]
```

2. @babel/plugin-transform-classes

同样的在项目中可能会使用 class 来面向对象编程，并且也经常会使用到继承来拓展基类的能力，例子如下：

```
class BaseTest {
  constructor(a) {
    this.a = a;
  }

  x() {}

  y() {}

  z() {}
}

class Test extends BaseTest {
  constructor(a) {
    super(a);
  }

  e(){
    super.x();
  }

  f() {}
}
```

当使用 babel plugin 中配置了默认的 @babel/plugin-transform-classes 时，该代码会编译如下代码：

```
"use strict";

```

```
function _get() { if (typeof Reflect !== "undefined" && Reflect.get) { _get = Reflect.get; } else { _get = function _get(target, property, receiver) { var base = _superPropBase(target, property); if (!base) return; var desc = Object.getOwnPropertyDescriptor(base, property); if (desc.get) { return desc.get.call(arguments.length < 3 ? target : receiver); } return desc.value; }; } return _get.apply(this, arguments); }

function _superPropBase(object, property) { while (!Object.prototype.hasOwnProperty.call(object, property)) { object = _getPrototypeOf(object); if (object === null) break; } return object; }

function _inherits(subClass, superClass) { if (typeof superClass !== "function" && superClass !== null) { throw new TypeError("Super expression must either be null or a function"); } subClass.prototype = Object.create(superClass && superClass.prototype, { constructor: { value: subClass, writable: true, configurable: true } }); Object.defineProperty(subClass, "prototype", { writable: false }); if (superClass) _setPrototypeOf(subClass, superClass); }

function _setPrototypeOf(o, p) { _setPrototypeOf = Object.setPrototypeOf || function _setPrototypeOf(o, p) { o.__proto__ = p; return o; }; return _setPrototypeOf(o, p); }

function _createSuper(Derived) { var hasNativeReflectConstruct = _isNativeReflectConstruct(); return function _createSuperInternal() { var Super = _getPrototypeOf(Derived), result; if (hasNativeReflectConstruct) { var NewTarget = _getPrototypeOf(this).constructor; result = Reflect.construct(Super, arguments, NewTarget); } else { result = Super.apply(this, arguments); } return _possibleConstructorReturn(this, result); }; }

function _possibleConstructorReturn(self, call) { if (call && (typeof call === "object" || typeof call === "function")) { return call; } else if (call !== void 0) { throw new TypeError("Derived constructors may only return object or undefined"); } return _assertThisInitialized(self); }

function _assertThisInitialized(self) { if (self === void 0) { throw new ReferenceError("this hasn't been initialised - super() hasn't been called"); } return self; }

function _isNativeReflectConstruct() { if (typeof Reflect === "undefined" || !Reflect.construct) return false; if (Reflect.construct.sham) return false; if (typeof Proxy === "function") return true; try { Boolean.prototype.valueOf.call(Reflect.construct(Boolean, [], function () {})); return true; } catch (e) { return false; } }

function _getPrototypeOf(o) { _getPrototypeOf = Object.setPrototypeOf ? Object.getPrototypeOf : function _getPrototypeOf(o) { return o.__proto__ ||
```

```
Object.getPrototypeOf(o); }; return _getPrototypeOf(o); }

function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructor)) { throw new TypeError("Cannot call a class as a function"); } }

function _defineProperties(target, props) { for (var i = 0; i < props.length; i++) { var descriptor = props[i]; descriptor.enumerable = descriptor.enumerable || false; descriptor.configurable = true; if ("value" in descriptor) descriptor.writable = true; Object.defineProperty(target, descriptor.key, descriptor); } }

function _createClass(Constructor, protoProps, staticProps) { if (protoProps) _defineProperties(Constructor.prototype, protoProps); if (staticProps) _defineProperties(Constructor, staticProps); Object.defineProperty(Constructor, "prototype", { writable: false }); return Constructor; }

let BaseTest = /*#__PURE__*/function () {
  function BaseTest(a) {
    _classCallCheck(this, BaseTest);

    this.a = a;
  }

  _createClass(BaseTest, [{
    key: "x",
    value: function x() {}
  }, {
    key: "y",
    value: function y() {}
  }, {
    key: "z",
    value: function z() {}
  }]);

  return BaseTest;
}();

let Test = /*#__PURE__*/function (_BaseTest) {
  _inherits(Test, _BaseTest);

  var _super = _createSuper(Test);

  function Test(a) {
    _classCallCheck(this, Test);

    return _super.call(this, a);
```

```
  }

  _createClass(Test, [{
    key: "e",
    value: function e() {
      _get(_getPrototypeOf(Test.prototype), "x", this).call(this);
    }
  }, {
    key: "f",
    value: function f() {}
  }]);

  return Test;
}(BaseTest);

```

可以看出 babel 编译后的类继承还是比较复杂的，涉及了比较多的函数调用，我们使用 benchmark 分别来测试一下构造函数和实例方法调用的性能，同时测试一下构造10万个实例后内存上的开销，测试代码如下：

```

const Benchmark = require('benchmark');
const process = require('process');

const t = new Test();
const suite = new Benchmark.Suite();

suite
.on('complete', (event) => {
    console.log(String(event.target));
})
.add('new Test', () => {
    const t = new Test();
})
.run()
.add('t.e()', () => {
  t.e();
})
.run()

const arr = [];
const before = process.memoryUsage();
for (let i = 0; i < 100000; i++) {
  arr.push(new Test());
}
```

```
26 console.log(`10w Test heapUsed diff: ${(process.memoryUsage().heapUsed -
   before.heapUsed) / 1024 / 1024}MB`);
```

其测试结果如下：

```
1 new Test x 1,446,508 ops/sec ±1.21% (87 runs sampled)
2 t.e() x 41,960,280 ops/sec ±0.36% (93 runs sampled)
3 10w Test heapUsed diff: 26.5MB
```

如果不使用 @babel/plugin-transform-classes 则 babel 不会对 class 进行编译，其测试结果如下：

```
1 new Test x 171,730,493 ops/sec ±0.46% (92 runs sampled)
2 t.e() x 24,297,804 ops/sec ±0.21% (94 runs sampled)
3 10w Test heapUsed diff: 5.2MB
```

当然如果使用 @babel/plugin-transform-classes 并且配置为宽松模式，则 babel 会编译成一种简单的继承方式（复制原型链的方式），同样的进行测试后其结果如下：

```
1 new Test x 826,371,067 ops/sec ±1.68% (84 runs sampled)
2 t.e() x 833,356,353 ops/sec ±1.74% (87 runs sampled)
3 10w Test heapUsed diff: 5.2MB
```

根据结果可以看出，使用宽松模式编译后其运行的速度比前两者会快几倍甚至百倍，并且内存的开销也是最小的，那宽松模式和严格模式上有什么差别呢？这里笔者没有深入去查阅相关资料，目前知道的影响是在宽松模式一下，其**基类上的** `new.target` 是 `undefined` ，也欢迎大家在评论区讨论。

综上所述这里推荐配置如下：

```
1  "plugins": [
2    [
3      "@babel/plugin-transform-classes",
4      {
5        "loose": true
6      }
7    ]
8  ]
```

3. assumptions

在上面的链接中，可以发现 babel 在 7.13.0 之后新增了 assumptions 的配置，其取代了宽松模式的配置，便于更好的优化编译结果。这里就不再给出推荐配置了，建议大家动手尝试灵活配置。

TypeScript 编译优化

> 本内容运行环境为 node v14.16.0，benchmark 版本为 benchmark@2.1.4，typescript 版本为 typescript@4.6.4，webpack 版本为 webpack@5.72.0

同样的 typescript 也有非常多的配置项，不过好在大多数配置并不会对性能造成很大的影响，这里主要介绍 typescript 与 webpack 等编译工具结合使用后，将多文件编译成单文件引起的性能问题。

在项目中，我们通常会进行模块划分，将各个模块拆分为单独的文件，把相似的模块归类到同一个文件夹下，同时还会在对应文件夹下创建一个 index 文件，并将该目录下的全部模块进行一个导出，这样做既方便了不同模块间的引用方式，也方便了模块管理和摇树等等，简单例子如下：

```
// 目录结构
.
└── src
    ├── demo.ts
    └── lib
        ├── constants
        │   ├── number.ts
        │   └── index.ts
        └── index.ts
```

```ts
// src/lib/constants/number.ts
export const One: number = 1;

// src/lib/constants/index.ts
export * from 'number';

// src/lib/index.ts
export * from 'constants';

// demo.ts
import { One } from './lib';
```

```
12
13 function demo() {
14   for (let i = 0; i < 100; i++) {
15     if (i === One) {
16       // do something
17     }
18   }
19 }
20
21 // 性能测试代码
22 const Benchmark = require('benchmark');
23 const suite = new Benchmark.Suite();
24 suite
25 .on('complete', (event) => {
26     console.log(String(event.target));
27 })
28 .add('import benchmark test', demo)
29 .run();
30
```

假设我们使用 webpack 进行编译并只配置一个 ts-loader，同时修改 tsconfig.json 中的配置将 **compilerOptions.module** 配置为**非 esnext** 的参数，比如为 commonjs，那么当 demo.ts 作为入口文件，编译输出成单文件后，其内部每个导出的 index.ts 模块都会被编译成如下代码：

```
1 var __createBinding = (this && this.__createBinding) || (Object.create ?
  (function(o, m, k, k2) {
2     if (k2 === undefined) k2 = k;
3     var desc = Object.getOwnPropertyDescriptor(m, k);
4     if (!desc || ("get" in desc ? !m.__esModule : desc.writable ||
  desc.configurable)) {
5       desc = { enumerable: true, get: function() { return m[k]; } };
6     }
7     Object.defineProperty(o, k2, desc);
8 }) : (function(o, m, k, k2) {
9     if (k2 === undefined) k2 = k;
10    o[k2] = m[k];
11 }));
12 var __exportStar = (this && this.__exportStar) || function(m, exports) {
13     for (var p in m) if (p !== "default" &&
  !Object.prototype.hasOwnProperty.call(exports, p)) __createBinding(exports, m,
  p);
14 };
15 Object.defineProperty(exports, "__esModule", ({ value: true }));
```

可以看出**每一层的导出的内容都会被 getter 包裹一次**，那么在外部访问对应模块时是层级越深，走过的 getter 次数越多，从而增加了性能的开销，以上面内容为例其 benchmark 结果为：

```
import benchmark test x 874,452 ops/sec ±0.12% (95 runs sampled)
```

当 module 配置为 esnext 后，其 benchmark 结果为：

```
import benchmark test x 21,961,693 ops/sec ±1.48% (90 runs sampled)
```

可以看出在使用 esnext 的场景下性能快了 20 多倍。可能有读者会问如果就是要使用 commonjs 的导出方式还有办法优化吗？答案是肯定的，这里给出几种解法：

1. 修改引用路径，直接引导最内部的文件，降低 getter 的次数
2. 在使用的文件中定义一个变量将对应的值存储起来，如将 demo 修改为如下代码：

```
import { One } from './lib';
const SelfOne = One;

function demo() {
  for (let i = 0; i < 100; i++) {
    if (i === SelfOne) {
      // do something
    }
  }
}
```

3. 不使用 ts-loader，使用 @babel/preset-typescript + babel loader

JavaScript 逻辑优化

JavaScript 逻辑方面最好的优化手段还是通过 devtool 录制 performance 来进行性能分析，这里给出几个优化思路：

1. 当频繁的使用同一个数组进行查找内容时，如果**不需要考虑索引且该数组内容不重复**，可用 Set 代替其时间复杂度

```
// 优化前
```

```
const arr = ['A', 'B', 'C'];
function isIncludes(string) {
  return arr.includes(string);
}

// 优化后
const set = new Set(['A', 'B', 'C']);
function isIncludes(string) {
  return set.has(string);
}

```

2. 当 if else 特别多时一般会建议用 switch case，当然改用 switch case 后还有两种优化方案，一是把**容易匹配的 case 放在前面**，不容易匹配的放后面；二是用 Map/Object 的形式把每种 case 当做一个函数来处理

```
// 优化前
if (type === 'A') {
  // do something
} else if (type === 'B') {
  // do something
} else if (type === 'C') {
  // do something
} else {
  // do something
}

// 优化方案一
switch (type) {
// 命中率高的放前面
case 'C':
  // do something
  break;
// 命中率次高的放中间
case 'B':
  // do something
  break;
// 命中率低的放后面
case 'A':
  // do something
  break;
default:
  // do something
  break;
}
```

```
30
31 // 优化方案二
32 function A() {
33   // do something
34 }
35
36 function B() {
37   // do something
38 }
39
40 function C() {
41   // do something
42 }
43
44 function defaultFn() {
45   // do something
46 }
47
48 const map = { A, B, C };
49
50 if (map[type]) {
51   map[type]();
52 } else {
53   defaultFn();
54 }
55
```

3. 高频率使用的计算函数，如果**频繁的存-在重复的输入输出**时，可考虑使用缓存来减少计算，当然缓存也不能乱用，不然可能会产生大量的内存增长

```
1 // 优化前
2 const fibonacci = (n) => {
3   if (n === 1) return 1;
4   if (n === 2) return 1;
5   return fibonacci(n-1) + fibonacci(n-2);
6 };
7
8 // 优化后
9 import { memoize } from 'lodash-es';
10
11 const fibonacci = memoize((n) => {
12   if (n === 1) return 1;
13   if (n === 2) return 1;
14   return fibonacci(n-1) + fibonacci(n-2);
15 });
```

4. 当要进行数组合并，且原数组不需要保留时，用 push.apply 代替 concat，前者的时间复杂度是 O(n) ，而后者因为是**将数组A和数组B合并成一个新数组C**，所以时间复杂度是 O(m+n)，当然如果数组过长那么 push.apply 可能会引起爆栈，可通过 for + push 解决

```
// 优化前
arrA = arrA.concat(arrB);

// 优化后
arrA.push.apply(arrA, arrB);

// 当上述优化有溢出问题时可改用下面这种
for (let i = 0, len = arrB.length; i < len; i++) {
  arrA.push(arrB[i]);
}
```

5. 尽可能减少链式调用将逻辑放到一个函数内，一是可以减少调用栈的长度；二是可以减少一些链式调用上的隐式开销

```
function square(v) {
  return v * v;
}

function isLessThan5000(v) {
  return v < 5000;
}

// 优化前
arr.map(square).filter(isLessThan5000).reduce((prev, curr) => prev + curr, 0);

// 优化后
arr.reduce((prev, curr) => {
  curr = square(curr);
  if (isLessThan5000(curr)) {
    return prev + curr;
  }
  return prev;
}, 0);
```

6. 当要等待多个异步任务结束后完成某个工作时，如果这些异步任务之间无关联关系，用 Promise.all 代替一个个 await

```
// 优化前
await getPromise1();
await getPromise2();
// do something

// 优化后
await Promise.all([getPromise1(), getPromise2()]);
// do something

```

7. 当判断条件中使用与或运算符时，把性能好的放前面

```
// 优化前
if (check0n() || check01()) {
  // do something
}

// 优化后
if (check01() || check0n()) {
  // do something
}
```

Canvas 优化

由于笔者工作中主要与 canvas2d 打交道，所以这里的分享也主要是与 canvas2d 相关的：

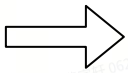
1. 当有 canvas 内容滚动或移动的需求时，如果本身 canvas 内容是**复杂的且非透明背景色**，则可以通过 **drawImage 自己**来减少绘制区域

需求：将下图内容向下滚动两个数字

0
1
2
3
4
5
6
7
8
9

1. 将红框部分绘制到 0 的位置

0
1
2
3
4
5
6
7
8
9



2
3
4
5
6
7
8
9
8
9

2. 绘制剩余部分内容

2
3
4
5
6
7
8
9
10
11

```
// 假设例子为垂直方向每 10px 展示一个数字从 0 开始
// 当前页面宽度 200 高度 100 向下滚动 20px

const width = 200;
const height = 100;
const offset = 20;

// 优化前
ctx.fillStyle = '#fff';
```

```
ctx.fillRect(0, 0, width, height); // 设置白色背景
ctx.fillStyle = '#000';
for (let i = 0, len = height / 10; i < len; i++) { // 每 10 px 绘制一个数字
  ctx.fillText(2 + i, width / 2, (i + 1) * 10);
}

// 优化后
ctx.drawImage(
  ctx.canvas,
  0, offset, 200, height - offset,
  0, 0, 200, height - offset
); // 绘制已有内容
ctx.fillStyle = '#fff';
ctx.fillRect(0, height - offset, width, offset); // 设置白色背景
ctx.fillStyle = '#000';
for (let i = 0, len = offset / 10; i < len; i++) { // 绘制剩余的数字
  ctx.fillText(10 + i, width / 2, (i + 1) * 10 + height - offset);
}

```

2. 如果在 canvas 中有绘制图标的需求，且图标本身是用 SVG 描述的，那么可以将 SVG 转成 Path2D 来，通过用 Path2D 绘制替代 drawImage 绘制
3. 减少 canvas2d 上下文的切换，尽可能保持相同上下文绘制完成后再切换，如需要交替展示红黄绿，可以先把红色部分全部绘制完，再绘制黄色以及绿色，而非每画一个区域切换一个颜色

React 优化

React 的优化个人认为是最困难的，常见的有减少不必要的 state 更新或通过一些 api 来减少 render 次数、非必需的组件懒加载、状态批量更新等等。它没有快速优化的手段，只能通过一些工具去逐步分析优化，这里就不做过多的描述了，简单提供几个分析工具的链接：

1. 官方提供的 React Profiler 工具
2. 开源的 why-did-you-render

另外再补充一个比较全面的关于 React 优化的文章：

https://juejin.cn/post/6935584878071119885

结语

笔者在工作中做过很多性能优化相关的工作，但一直以来都没有进行一些总结和分享，这次利用五一假期时间对之前的优化做了简单的梳理和总结，算是完成了写一篇分享的小目标。同时也希望这篇文章对大家有帮助，可以拓宽日常工作中的优化思路。

如果您对文章有疑问或者有更多的优化技巧，欢迎评论交流。

另外关于内存相关的优化可以看这篇👉 如何减少 JavaScript 运行时内存开销

同时也欢迎活水加入表格团队一起做更有挑战的事情👉 字节很缺前端！-飞书在线表格&幻灯片（活水|自荐也可）